卒業研究論文

エントリの再利用性を考慮した
共有キャッシュの動的分割手法

指導教員　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 23 年度入学　23115115 番

濵西 夏生

平成 27 年 2 月 9 日

エントリの再利用性を考慮した
共有キャッシュの動的分割手法

濵西 夏生

内容梗概

これまでゲート遅延に対し，配線遅延が相対的に増大してきたことにより，配線遅延がプロセッサ性能に与える影響が大きくなってきた．このため，配線遅延を隠蔽するキャッシュメモリの重要性が高まってきた．この一方で，集積回路の微細化に伴うリーク電流の増大による消費電力及び発熱量の増大といった問題から，プロセッサの動作周波数の向上は困難になってきている．このことから，現在では消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力を向上させるため，単一チップ上に複数のコアを搭載したマルチコアプロセッサが広く普及している．このようなマルチコアプロセッサでは，キャッシュを有効活用するため複数のコアがラストレベルキャッシュを共有する場合が多い．ここで，複数のコアによって共有されるキャッシュ上では，あるコアで動作しているプロセスによって確保されたキャッシュエントリが，他のコアで動作しているプロセスに追い出される干渉が発生し，キャッシュヒット率が著しく低下する場合がある．そこで，こうした干渉の発生を抑制する手法として，各コアに占有領域を与えるキャッシュ分割手法が提案されている．

このようなキャッシュ分割手法の 1 つである RWP（Read Write Partitioning）では，キャッシュエントリの再利用性を考慮してキャッシュ領域を分割する．この RWP は，エントリを再参照する方法を用途に応じて 2 種類に分類し，それに合わせてキャッシュ領域を 2 分割するのみで，キャッシュミス発生回数の抑制を実現できるという利点を持つ．しかし，RWP では干渉による性能低下については深く考察されていない．そこで，本研究ではこの RWP に対して，各コアに占有領域を与えることで干渉による性能低下を抑制する手法を提案する．提案した手法の有効性を検証するため，既存の RWP に提案手法を実装し，SPEC CPU 2006 を用いてシミュレーションにより評価した．その結果，既存の RWP と比較して，共有キャッシュに対する読み出しアクセスのヒット率を最大 46 %，平均 29 %向上できることが確認できた．

エントリの再利用性を考慮した
共有キャッシュの動的分割手法

目次

1　はじめに

プロセッサの高速化は，主に集積回路の微細化による高クロック化によって実現されてきた．しかしながら，集積回路の配線遅延がゲート遅延に対して相対的に増大し，配線遅延がプロセッサ性能に与える影響が大きくなってきたことにより，高クロック化のみではプロセッサを高速化させることが難しくなった．このため，配線遅延を隠蔽するための手段としてキャッシュメモリが着目され，その高性能化手法が広く研究されてきた．キャッシュメモリを高性能化させるための手法は多岐に渡り，その動作をフルアソシアティブに近づける手法 [1, 2], 追い出しの優先度を変える手法 [3, 4, 5, 6]，データをプリフェッチする手法 [7, 8] など様々な手法が存在する．

この一方で，集積回路の微細化に伴う消費電力及び，発熱量の増大といった問題から，プロセッサのクロック周波数の向上自体も困難になってきている．このような中で，クロック周波数向上以外のプロセッサ高速化手法として，SIMD やスーパスカラのような命令間の並列性に着目した手法が利用されるようになってきた．しかしながら，プログラム中の命令間の並列性には限界があり，命令間の並列性に依存した性能向上も頭打ちになりつつある．このため，現在ではプロセッサあたりの処理能力を向上させるため，単一チップ上に複数のコアを搭載したマルチコアプロセッサが広く普及している．このようなマルチコアプロセッサでは，各コアの動作周波数を低く抑え，消費電力及び発熱量の問題の解決を図っている．これにより，配線遅延の相対的な増大がプロセッサ性能に与える影響は抑えられつつあるが，その影響は依然として大きく，キャッシュ性能を向上させる重要性も依然として大きい．

さて，このようなマルチコアプロセッサでは，キャッシュメモリを有効活用するために複数のコアがキャッシュの一部を共有する場合が多い．ここで，複数のコアに共有されるキャッシュ上では，あるコアで動作しているプロセスによって確保されたキャッシュエントリが，他のコアで動作しているプロセスに追い出される**干渉**が発生し，プロセッサの性能が著しく低下してしまう場合がある．そこで，こうした干渉の発生を抑制する手法として，各コアが利用可能なキャッシュ領域を制限するキャッシュ分割手法が提案されている．

このようなキャッシュ分割手法の 1 つである **RWP（Read Write Partitioning）** [9] では，キャッシュエントリの再利用性を考慮してキャッシュ領域を分割する．この RWP は，エントリを再参照する方法を用途に応じて 2 種類に分類し，それに合わせてキャッシュ領域を 2 分割するのみで，キャッシュミス発生回数の抑制を実現できるという利点を持つ．しかし，RWP では干渉による性能低下については深く考察されていない．そこで，

本研究ではRWPに対して，分割したキャッシュ領域を占有領域として各コアに与えることで干渉による性能低下を抑制する手法を提案する．以下，2章では，本研究で着目する手法であるRWPを説明するにあたり，従来のキャッシュ分割手法とRWPとの着眼点の違いについて述べ，その動作について説明する．3章および4章ではRWPの抱える問題点を述べ，本論文で提案する手法について説明する．5章で提案手法の性能を評価し，最後の6章において結論を述べる．

2　研究背景

本研究で着目するRWPは，従来のキャッシュ分割手法とは異なる着眼点からキャッシュの高性能化を図る手法である．本章では，まずRWPの着眼点が従来のキャッシュ分割手法とどう異なっているかについて説明する．その後，RWPの具体的な動作と実装について説明する．

2.1　RWPの着眼点

RWPは，ストア命令とロード命令によるキャッシュメモリに対するアクセスがミスした時の，プロセッサの挙動の違いに着目した手法である．ストア命令により，キャッシュメモリに対して書き込みアクセスが発生し，そのアクセスがミスした場合，キャッシュメモリへ書き込まれる予定であったデータは，あらかじめ用意されたバッファに書き込まれる．そのため，実際にメモリからキャッシュへ書き換えたいデータが読み出されるまでの間，プロセッサはこのバッファに記憶されているデータを利用することで，その後の実行を継続できる．つまり，メモリからキャッシュへデータが読み出されるまでの間，プロセッサは実行を待機する必要は無いため，書き込みアクセスのミスによるメモリアクセス時間は隠蔽される．一方，ロード命令により，キャッシュメモリに対して読み出しアクセスが発生し，そのアクセスがミスした場合は，データをメモリからキャッシュへ読み出す必要がある．このため，その読み出しが完了するまでの間，プロセッサは実行を待機しなければならない．よって，書き込みアクセスがミスした場合とは異なり，メモリアクセス時間は隠蔽されない．したがって，キャッシュメモリに対するアクセスにおいては読み出しアクセスのミスの方が，書き込みアクセスのミスよりもプロセッサ性能へ与える影響が大きい．そのため，書き込みアクセスよりも読み出しアクセス対象のキャッシュエントリを優先するよう管理するべきであると考えられる．ここで，そのように管理した場合と，これらを区別せずに追い出しアルゴリズムとしてLRU（Least Recently Used）を用いてキャッシュエントリを管理した場合とで，読み出しアクセスのミス回数，つまりメモリア

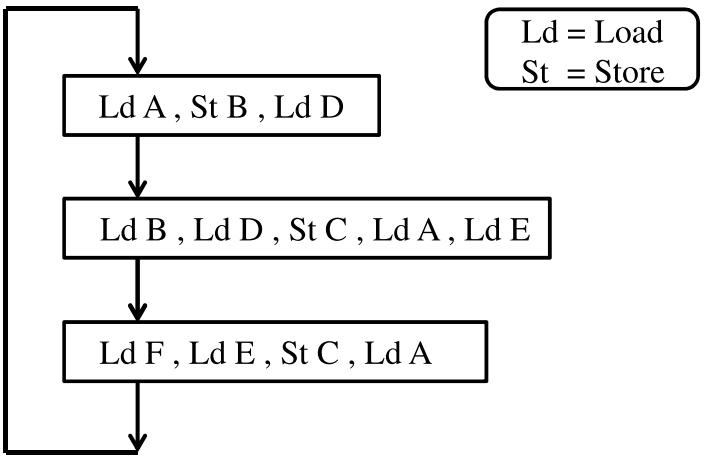
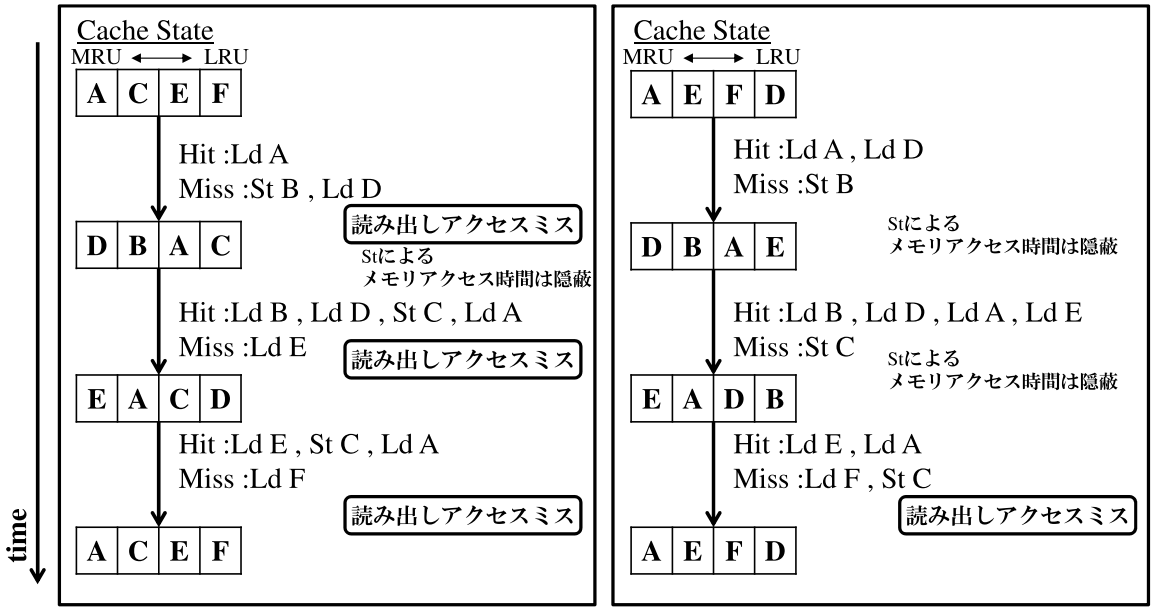


図 1: あるループ処理中のメモリアクセスパターン

クセスが完了するまでの間，プロセッサが処理を待機するという状況が発生する回数にどの程度の差が生まれるか，あるループ処理中のメモリアクセスパターンを示した**図 1** と，そのループ処理を実行する際のキャッシュメモリの状態の遷移を示した**図 2** を用いて説明する．なお，説明に用いるキャッシュは 4 ウェイのセットアソシアティブキャッシュとする．また，図 1 のメモリアクセス命令は，このキャッシュを LRU を用いて管理した場合に，読み出しアクセスのミスが発生するまでの一連のアクセス，または発生した後の一連のアクセスをブロックで囲みグループ化して示している．すなわち，キャッシュに対するアクセスがミスするロード命令 1 つとその命令の前もしくは後に続く複数のメモリアクセス命令が含まれる命令列を，1 つの命令グループとしてまとめて示しており，図 2 はこのグループ毎にキャッシュメモリの状態遷移を示している．

図 1 に示すメモリアクセスパターンをもつループ処理中では，アドレス A，D，E，F 上のデータはロード命令のみによってアクセスされる．また，アドレス C 上のデータはストア命令のみによってアクセスされる．そして，アドレス B 上のデータはロード，ストア命令の両方によってアクセスされる．図 2 は，このループ処理の 2 回目以降の実行過程を示しており，(a) が LRU を用いてキャッシュエントリを管理した場合，(b) が書き込みアクセスよりも読み出しアクセス対象のキャッシュエントリを優先して管理した場合を示している．それぞれの管理方法の違いから，初期参照ミスの回数が異なるため，図 2 で示すキャッシュの初期状態は，(a)，(b) のそれぞれで異なっている．また，各キャッシュ状態は一番左にあるエントリが，参照時刻の最も新しいエントリ，つまり MRU（Most Recently Used）なエントリであることを示しており，右にいくにつれ古いエントリであることを示している．図 2 より，(a) の場合は，ループの 1 回の処理で，読み出しアクセス



(a) LRUを用いた場合　　(b) 特徴に着目した場合

図 2: キャッシュメモリの状態遷移

のミスが必ず 3 回発生することがわかる．これはメモリアクセスパターンを考慮せずに，単純に LRU を用いてキャッシュエントリを用いて管理したことに起因する．一方で，(b) の場合は，ロード命令によってアクセスされるアドレス A，B，D，E，F 上のデータと，そうでないアドレス C 上のデータを区別する．そして先に述べたように，書き込みアクセスよりも，読み出しアクセス対象のキャッシュエントリを優先して管理した場合の影響を確認するため，ここでは，ストア命令によってのみアクセスされるアドレス C 上のデータをキャッシュしない場合を想定する．すると，図 2 のように，このループの 1 回の繰り返し処理において，読み出しアクセスのミス回数が 1 回に抑えられることが分かる．このように，読み出しアクセスのミスと書き込みアクセスのミスがプロセッサ性能へ与える影響の違いに着目することで，読み出しアクセスのミスが発生する回数，つまり，メモリアクセスが完了するまで，プロセッサが処理を待機するという状況が発生する回数を抑制できる場合が存在することが分かる．この特徴に着目し，RWP ではキャッシュの高性能化を図っている．

2.2　RWP の動作と実装

本節では，本研究で対象とする RWP の概要について述べ，RWP を実現するために必要となるハードウェア機構を説明する．そして，それらのハードウェア機構を用いた RWP

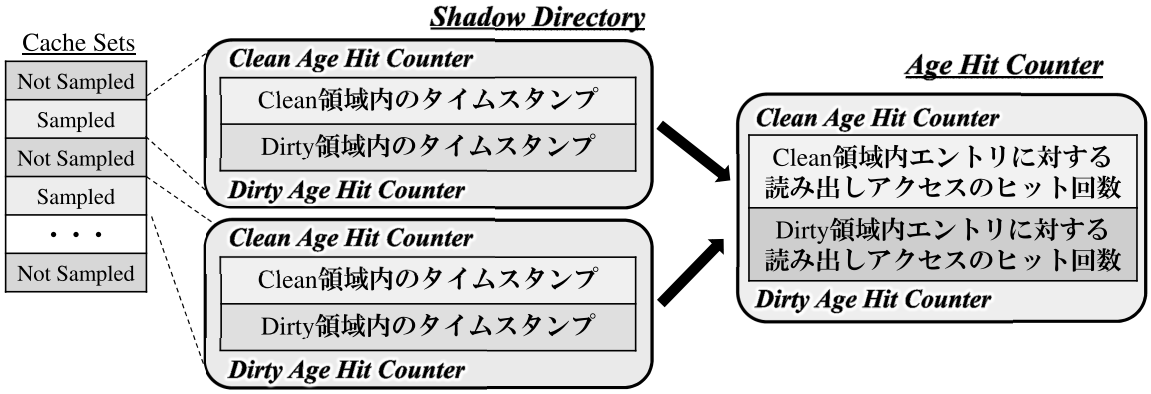


図 3: Shadow Directory と Age Hit Counter

の動作を説明する，

2.2.1 RWP の概要

先に述べたように RWP は読み出しアクセスのミスと書き込みアクセスのミスがプロセッサ性能へ与える影響の違いに着目する．そこで，RWP ではセットアソシアティブキャッシュにおいて各セットを，読み出しアクセスによってのみ再参照されるデータが配置される **Clean 領域**と，読み出しと書き込みアクセスの両方によって再参照されるデータが配置される **Dirty 領域**の 2 つの領域に分割し，これら 2 つの領域サイズを動的に変化させることで，キャッシュメモリに対する読み出しアクセスのミスが発生する回数を抑制する．この 2 分割を実現するために RWP では，各キャッシュエントリに，当該キャッシュエントリの属する領域が Clean 領域か，Dirty 領域かを示すビット（Dirty ビット）を追加する．これにより RWP はキャッシュ領域を論理的に分割することができる．また，これら 2 つの領域サイズを動的に変化させ，読み出しアクセスのミスが発生する回数を抑制できる内訳となるよう調整するために，RWP では，Dirty ビットとは別に以下の 2 つの追加ハードウェアを用いる．

- **Shadow Directory**
  各セット内でのエントリのタイムスタンプを管理
- **Age Hit Counter**
  Shadow Directory で管理している各タイムスタンプを持つエントリに対する読み出しアクセスのヒット回数を記憶

これらを共有キャッシュ上に実装した様子を**図 3** に示す．なお RWP は共有キャッシュにのみ実装される．Shadow Directory は 1 つのセットにつき 1 つ実装され，各エントリ

のセット内におけるタイムスタンプ情報を保持する．RWP は追い出しアルゴリズムとして LRU を採用しており，これはこの Shadow Directory の保持するタイムスタンプを用いて実現されている．各 Shadow Directory は，タイムスタンプを Clean/Dirty の領域毎に管理するために，Clean 領域のタイムスタンプを記憶する Clean Shadow Directory と Dirty 領域のタイムスタンプを記憶する Dirty Shadow Directory から構成される．これにより Shadow Directory を用いることで，各セット内において Clean/Dirty のそれぞれの領域内で最も長い間アクセスされていないエントリを見つけることができ，LRU によるキャッシュエントリの追い出しが可能となる．また，このように Clean/Dirty の領域毎に別々のタイムスタンプを用いて管理することにより，Age Hit Counter が Clean/Dirty 領域に属するエントリに対する読み出しアクセスのヒット回数を，そのエントリの保持するタイムスタンプ毎に記憶することもできる．この Age Hit Counter はキャッシュセット全体に対して 1 つ実装され，Clean 領域に属するエントリに対する読み出しアクセスのヒット回数を記憶する Clean Age Hit Counter と Dirty 領域に属するエントリに対する読み出しアクセスのヒット回数を記憶する Dirty Age Hit Counter により構成される．そして，これら 2 つの Age Hit Counter は，エントリに対する読み出しアクセスのヒット回数を，タイムスタンプ毎に保持する．したがって，Shadow Directory では Clean/Dirty の領域毎に別々のタイムスタンプが用いられる．ここで，RWP を実装した 8 ウェイセットアソシアティブキャッシュの構成を示している**図 4** を用いて，Shadow Directory と Age Hit Counter の構成をより具体的に説明する．なお図 4 の大文字の C，D はそれぞれ，そのエントリが属している領域が Clean 領域，Dirty 領域であることを示している．この図において，Shadow Directory が持つタイムスタンプは 1 から 8 の数字で表現され，1 が MRU なエントリであることを示し，数字が大きくなるにつれ各エントリは順に古いエントリであることを示している．また，Age Hit Counter 内の一番左のカラムは MRU なウェイに対する読み出しアクセスのヒット回数を記憶するエントリであり，そこから右方向へ，新しいウェイから順に，そのウェイに対する読み出しアクセスのヒット回数を記憶しているカラムが並んでいる．そして，Age Hit Counter の各カラムの way フィールドの値は，そのカラムの位置に対応するタイムスタンプを持つ実際のキャッシュエントリの way 番号を示している．例えば，図 4 の way6 は Clean な領域に属するエントリであり，そのタイムスタンプが 4 であるので，Clean 領域内で 3 番目に古いキャッシュエントリであることが分かる．したがって，Clean Age Hit Counter 内の左から 3 番目のカラムの way フィールドの値は 6 となり，もし way6 のキャッシュエントリに対して読み出しアクセスがヒットすると，このカラムの値をインクリメントする．そして，RWP では，こ

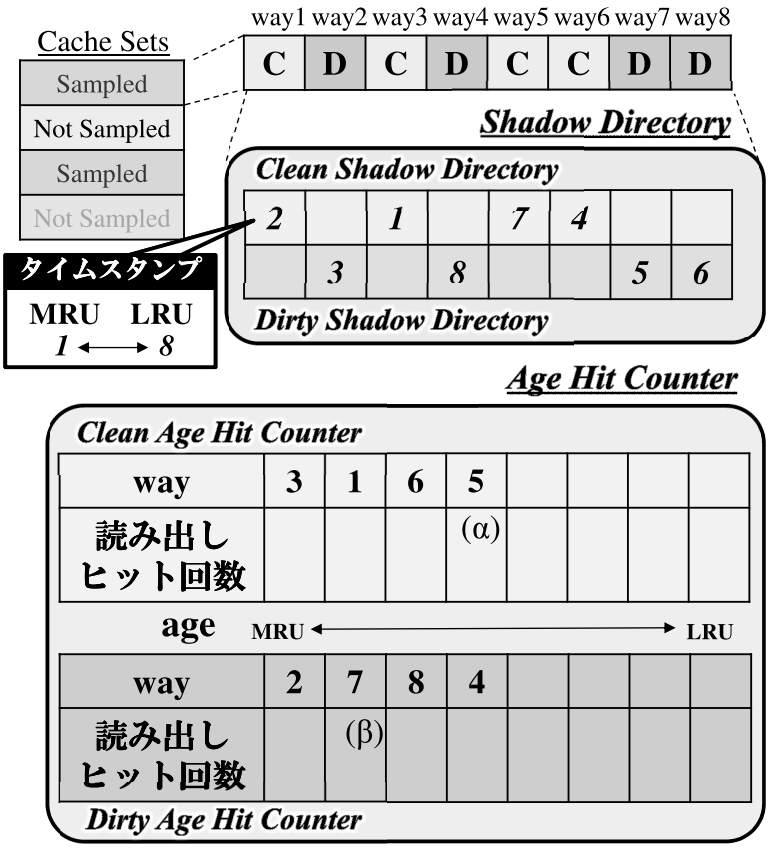


図 4: RWP を実装した 8way セットアソシアティブキャッシュの状態例

の Age Hit Counter の保持している値に基づき Clean/Dirty のどちらの領域を増大させると読み出しアクセスのヒット回数が増加するかを予測する．この予測方法に関しては，2.2.3 項で後述する．なお，Shadow Directory は DSS（Dynamic Set Sampling）[10] に基づき，一部のキャッシュセットを管理することで，キャッシュエントリ全体の再利用性を十分に考慮できると考えられるため，全てのキャッシュセットではなく，全キャッシュセット中の 32 セットにのみ実装される．図 3，図 4 では，Shadow Directory を実装しているセットを Sampled，そうでないセットを Not Sampled として表している．RWP では，以上のような 2 種類の追加ハードウェアを用いることで，Clean/Dirty のどちらの領域からエントリを追い出すと，読み出しアクセスのミスが発生する回数を抑制することができるかを判断する．

2.2.2 動作説明

本節では，RWP の動作を，図 4 に示す例を用いて，キャッシュへのアクセスがヒット，つまりキャッシュヒットした場合の動作と，キャッシュへのアクセスがミス，つまりキャッ

シュミスした場合にわけて説明する．なお，これ以降の説明で用いる全てのアクセスは図4に示した状態のキャッシュに対するアクセスであるとする．

まず, キャッシュヒットした場合の動作について説明する．この場合, RWP は Clean/Dirty 領域サイズを直接調整することなく，各エントリに対する読み出しアクセスのヒット回数を記憶する動作のみ行う．例えば，way7 のエントリ，すなわち Dirty 領域に対してアクセス要求があった場合，そのアクセスが書き込みであれば，LRU に基づいたキャッシュエントリの追い出しを実現するために，Shadow Directory のタイムスタンプを way7 のエントリが最も新しくなるように更新する．一方，そのアクセスが読み出しであれば，Shadow Directory の値の更新と同時に Age Hit Counter の値も更新する．これは，Shadow Directory が実装されているセット内の各エントリに対する読み出しアクセスのヒット回数をタイムスタンプ毎に記憶するためである．したがって，この場合は way7 のエントリが Dirty 領域内で 2 番目に古いエントリであるため，Dirty Age Hit Counter 内の左から 2 番目の値（図 4 中 (β) で示したカラムの値）がインクリメントされる．次に，例えば，way5 のエントリ，すなわち Clean 領域に対して，アクセス要求があった場合，そのアクセスが書き込みであれば，way5 のエントリが属する領域は Clean 領域から Dirty 領域に変化する．よって，way5 のエントリを管理する Shadow Directory の情報を Dirty に変更する．その後，タイムスタンプを way5 のエントリが最も新しくなるように更新する．一方，そのアクセスが読み出しであれば，Dirty 領域へのアクセスの時と同様に，読み出しアクセスのヒット回数をタイムスタンプ毎に記憶しておくために Clean Age Hit Counter の値をインクリメントする．つまりこの場合は，way5 のエントリが Clean 領域内で 4 番目に古いエントリであるため，Clean Age Hit Counter 内の左から 4 番目の値（図 4 中 (α) で示されたカラムの値）がインクリメントされる．その後，Shadow Directory のタイムスタンプを way5 のエントリが最も新しくなるように更新する．キャッシュヒットした場合は，以上で述べたように動作する．

次に，キャッシュミスした場合の動作について説明する．この場合はエントリを追い出す必要がある．そこでまず，Clean/Dirty のどちらの領域からエントリを追い出すかを決定する．エントリを追い出した領域のサイズは小さくなるため，領域サイズが大きすぎる領域からエントリを追い出すべきである．この判断は，Age Hit Counter に記憶してある，各エントリに対する読み出しアクセスのヒット回数に基づき，最も読み出しアクセスのヒット回数が多くなると予測された Dirty 領域サイズと，アクセスがミスした時点のアクセス先セットにおける Dirty 領域サイズとを比較することで行う．なお，この最も読み出しアクセスのヒット回数が多くなるような Dirty 領域サイズの算出方法については 2.2.3

項で後述する．この比較結果は，次の 3 つのいずれかとなる．

1. アクセスがミスした時点の Dirty 領域サイズが予測した Dirty 領域サイズより大きい．
2. アクセスがミスした時点の Dirty 領域サイズが予測した Dirty 領域サイズより小さい．
3. アクセスがミスした時点の Dirty 領域サイズが予測した Dirty 領域サイズと等しい．

1 番目の場合は，アクセス先セットにおける Dirty 領域サイズが大きすぎることを意味する．よって，Dirty 領域サイズを縮小するために，Dirty 領域からエントリを追い出す．2 番目の場合は，アクセス先セットにおける Dirty 領域サイズが小さすぎることを意味する．よって，Dirty 領域サイズを拡大するために，Clean 領域からエントリを追い出す．3 番目の場合は，アクセス先セットにおける Dirty 領域サイズを変更する必要がないことを意味する．この場合は，キャッシュミスを発生させたアクセスの種類により，Clean/Dirty のどちらの領域からエントリを追い出すかを決定する．つまり，もしそのアクセスが書き込みであれば Dirty 領域から，読み出しであれば Clean 領域からエントリを追い出す．このようにして，どちらの領域からエントリを追い出すかを決定した後，LRU に基づきその選択された領域からタイムスタンプの最も古いエントリを追い出す．以上で述べたように，キャッシュミスした際には，キャッシュヒット時に記憶した情報を用いて予測した Dirty 領域サイズと，アクセスがミスした時点のアクセス先セットにおける Dirty 領域サイズとを比較し，追い出すエントリを決定する．RWP は，これらの動作により，読み出しにヒットするエントリをキャッシュ上に多く残すことで，読み出しアクセスのミスが発生する回数を抑制する．

2.2.3 Clean/Dirty 領域内訳サイズの予測方法

2.2.2 項で述べたように，RWP ではキャッシュミスが発生した場合に，Age Hit Counter の値に基づいて最も読み出しアクセスのヒット回数が多くなると予測した Dirty 領域サイズを用いて，どちらの領域からエントリを追い出すかを決定する．本節では，この Dirty 領域サイズの算出方法について述べる．なお本論文では，この Dirty 領域サイズとそれに応じた Clean 領域サイズとを合わせて，予測 Clean/Dirty 内訳サイズと呼ぶ．

この算出にはキャッシュメモリの持つある特性を用いる．それは **Stack Property**[11] と呼ばれるものである．これは，追い出しアルゴリズムとして LRU を用いてキャッシュメモリを管理した場合，キャッシュメモリのウェイ数が n の場合にヒットするアクセス要求は，ウェイ数が n 以上である場合にも，必ずヒットするという特性である．RWP の目的は，キャッシュメモリに対する読み出しアクセスのミス回数を抑制することであり，これは読み出しアクセスのヒット回数を増加させることで実現できる．つまり，読み出しアクセスのヒット回数が最も多くなるように Clean/Dirty 領域の分割を行うことで，この

目的は達成できる．ここで，Age Hit Counter には各エントリに対する読み出しアクセスのヒット回数が Clean/Dirty 領域毎に保持される，つまり，Stack Property により，Age Hit Counter の値から，どちらかの領域サイズを 1 ウェイ増やした際に増加する読み出しアクセスのヒット回数を求めることが可能である．

ここで，エントリに対する読み出しアクセスが複数回ヒットした後の Age Hit Counter の状態と，予測 Clean/Dirty 内訳サイズを算出する過程を示した**図 5** を用いて，実際の予測過程を説明する．まず，Clean Age Hit Counter と Dirty Age Hit Counter から領域サイズ毎に，その領域サイズ以下のヒット回数の合計値，つまり読み出しアクセスのヒット回数の和を求める．そして，それらの和が最大となる時の，領域分割の内訳サイズを求め，それを予測 Clean/Dirty 内訳サイズとする．つまり，Age Hit Counter が図 5 に示す状態の場合は，Clean 領域サイズが 5 ウェイ以下の Clean Age Hit Counter の値の合計値 70 と，Dirty 領域サイズが 3 ウェイ以下の Dirty Age Hit Counter の値の合計値 55 との和である 125 が，全内訳中で最大となるため，それぞれの領域サイズ 5 ウェイと 3 ウェイが，図 5 の状態にある Age Hit Counter から算出される予測 Clean/Dirty 内訳サイズとなる．RWP では，このようにして Age Hit Counter から最も読み出しアクセスのヒット回数が多くなるような Clean/Dirty の内訳サイズを予測する．そしてこの場合，アクセスがミスした時点の Dirty 領域サイズ 4 ウェイのほうが予測した Dirty 領域サイズ 3 ウェイより大きいため，2.2.2 項で述べたように Dirty 領域からエントリを追い出す．なお，プログラムの実行フェーズに応じた予測を可能とするため，Age Hit Counter の値は一定の間隔でリセットされる．読み出しアクセスのヒット回数や間隔は，アクセス要求を出すプロセスの実行フェーズにより異なるため，こうすることで，最も読み出しアクセスのヒット回数が多くなる Clean/Dirty 領域の内訳サイズを実行フェーズに適した形で求めることが可能となる．

3　RWP の問題点とその解決方法

本章では，まず 2.2 節で説明した RWP の抱える問題点を示す．そして，その問題に対する解決方法を提示し，それをどのように RWP に適用するかを述べる．

3.1　RWP の問題点

RWP では，2.2 節で述べたように動作することで，読み出しアクセスのミス回数を抑制することを実現している．しかし，RWP は干渉による性能低下の解決に直接結びついてはいない．干渉は，マルチコア環境において共有キャッシュの性能を低下させてしまう

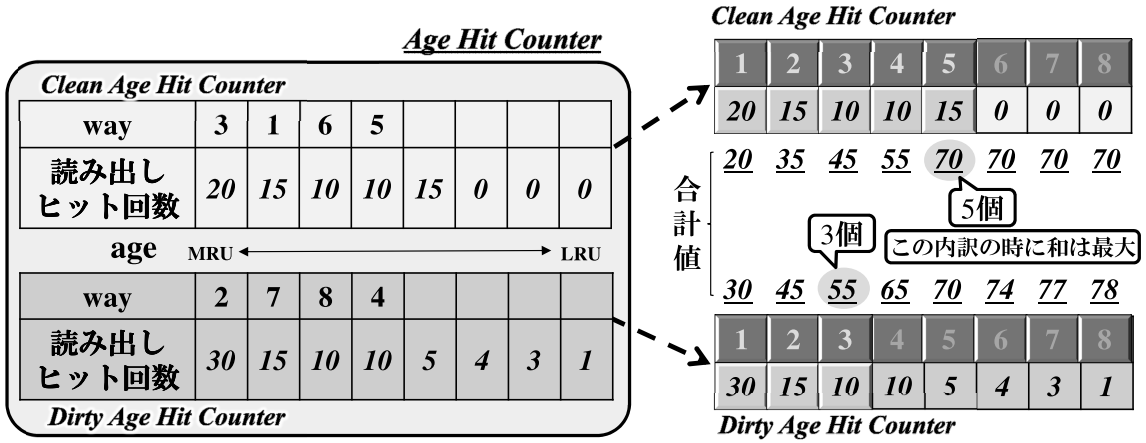


図5: 複数回読み出しアクセスがヒットした後の Age Hit Counter 及び，予測 Clean/Dirty 内訳サイズの算出過程

原因として一般に知られており，この問題を無視することはできない．

例えば，一度しか参照されないようなアドレスへのアクセスが多発するストリーミング処理を実行するプロセスがコア A 上で，Dirty 領域に対する読み出しアクセスが多いプロセスがコア B 上で同時に動作している場合の RWP の動作を考える．この場合，Dirty 領域サイズが拡大されると，コア B 上で動作しているプロセスによる読み出しアクセスのヒット回数は増加するため，本来は読み出しアクセスのミスによる性能低下を抑制できるはずである．しかし，実際は，コア A 上で動作しているプロセスによる，頻繁なキャッシュエントリの確保に伴い，コア B 上で動作するプロセスによるアクセスにより再参照されるエントリはキャッシュメモリから追い出されてしまう．このため，読み出しアクセスのミスによる性能低下を抑制することはできない．このように，RWP では干渉の発生による性能低下に対処できておらず，キャッシュ領域を Clean/Dirty の 2 つの領域に分割するだけでは性能向上に限界があることが分かる．

また, RWP では最も読み出しアクセスのヒット回数が多くなると予測される Clean/Dirty 領域の内訳サイズは動的に変化するため，プロセスの実行フェーズに応じた Clean/Dirty 領域の内訳サイズを求めることは可能となっている．しかし，読み出しアクセスのヒット回数を記憶する Age Hit Counter が 1 つのみであるため，予測 Clean/Dirty 内訳サイズがキャッシュ上の全セットで一意に決定されてしまう．このことを RWP を実装した様子を表している**図 6** を用いて説明する．この図は，RWP を実装した 8 ウェイセットアソシアティブキャッシュが複数のコアによって共有されており，Age Hit Counter の値に基づい

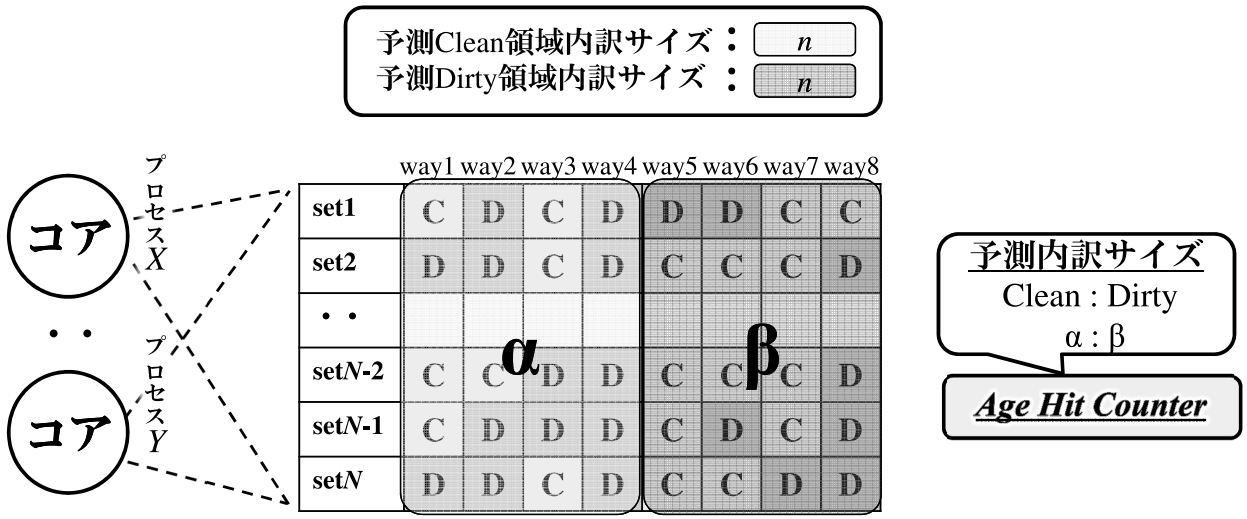


図 6: RWP を実装した様子

て算出した予測 Clean/Dirty 内訳サイズが $\alpha : \beta$ であることを示している．そして，図のようにこの $\alpha : \beta$ という 1 つの予測 Clean/Dirty 内訳サイズに，キャッシュセット全体の Clean/Dirty 領域の内訳サイズを近づけるように RWP は動作する．しかし，キャッシュセットを共有している各コア上ではそれぞれ異なるプロセスが動作しているため，RWP では複数のプロセスが同時に動作している際に各プロセスの特徴に応じた Clean/Dirty 領域の分割を行うことができていないと考えられる．例えば，Clean 領域に属するエントリに対する読み出しアクセスが頻繁に発生するプロセスと，Dirty 領域に属するエントリに対する読み出しアクセスが頻繁に発生するプロセスとが同時に動作していた場合，それぞれのプロセスが必要とする Clean/Dirty 領域のサイズは明らかに異なる．前者のプロセスでは，そのプロセスが引き起こす読み出しアクセスのミス回数を抑制するために，Clean 領域のサイズを大きくするべきであると考えられる．逆に，後者のプロセスでは，プロセスが引き起こす読み出しアクセスのミス回数を抑制するためには，Dirty 領域のサイズを大きくするべきであると考えられる．しかし，このようにプロセス毎に必要とされる Clean/Dirty 領域のサイズが異なる場合であっても，RWP では一意に決定された予測 Clean/Dirty 内訳サイズを用いてしまい，プロセス毎の特徴に応じた Clean/Dirty 領域の分割を行えない．そこで，本研究ではこの問題と先に述べた干渉による性能低下を回避できていないという問題の解決を図る手法を提案する．

3.2 提案手法と期待される効果

3.1 節で示した問題である干渉は，各コアに占有領域を与えることで解決でき，それを実

現する手法がこれまでに数多く提案されてきた．そこで，本研究でも同様の手法により，この問題の解決を図る．また，各コアに占有領域を与える手法の実現方法によっては，その占有領域毎に Clean/Dirty 領域の内訳サイズを設定することも可能となり，3.1 節で示した 2 つ目の問題も同時に解決できると考えられる．

ここで，各コアが占有領域を保持することを実現する代表的な方法として，以下の 2 つがある．

- ウェイ方向への分割 [12, 13]
- セット方向への分割 [14]

本研究では，これらの実現方法のうち，低コスト，すなわち少ないハードウェアの追加により実現できると考えられるうえ，占有領域毎に Clean/Dirty 領域の内訳サイズを設定することも可能になるセット方向への分割方法を採用する．

この，セット方向への分割により，各コアがアクセス可能なキャッシュセットを限定することができ，各コアは共有キャッシュ上に占有領域を持つことが可能となる．ここで，プロセス毎に必要とするキャッシュサイズは異なること，また，プロセスが必要とするキャッシュサイズは実行フェーズに応じて変化することが一般に知られている．したがって，より多くのキャッシュサイズを必要とするプロセスを動作させているコアが，より多くの占有領域を保持することにより，そのコアで動作しているプロセスが引き起こすキャッシュミスの発生回数を抑制できる．このため，提案手法では各コアが保持する占有領域サイズ，つまり占有セットサイズを動的に調整できるようにすることを目指す．また，このようにセット方向へキャッシュ領域を分割する方法を用いて，各コアに占有領域を与える手法を実現することにより，占有領域毎にウェイ方向への分割である Clean/Dirty 領域の分割を行うことが可能となる．したがって，各占有領域に異なる Clean/Dirty 領域の内訳サイズを設定することで，プロセス毎の特徴に応じた Clean/Dirty 領域の分割が可能となり，更なるキャッシュの高性能化が達成できると考えられる．このため，提案手法では動的にサイズが変化する占有領域毎に，異なる予測 Clean/Dirty 内訳サイズを算出することを目指す．

3.3 RWP への適用

3.2 節で述べた，セット方向へキャッシュ領域を分割する手法を RWP に組み合わせた提案手法について説明する．この手法により，各コアが占有領域，つまり占有セットを持つことが可能となる．実際に各コアが占有セットを保持している様子を**図 7** に示す．

この図では，コア s の保持する占有セットサイズが 2 セット，コア t の保持する占有セッ

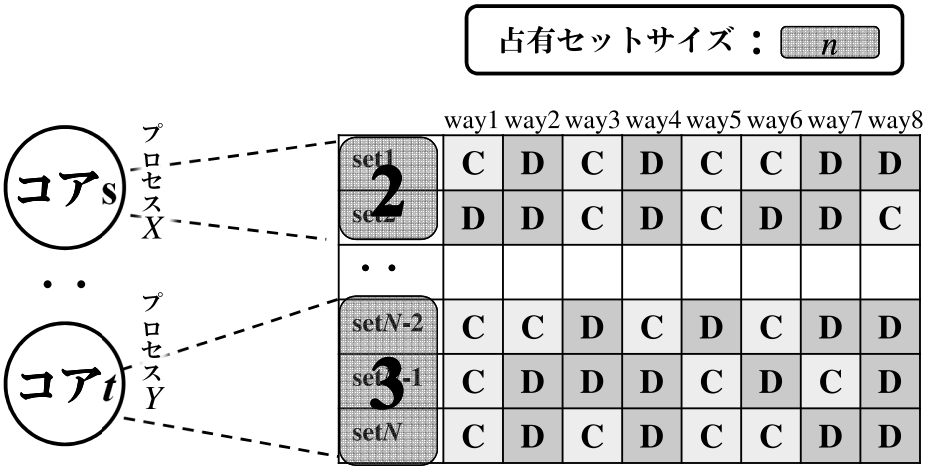


図 7: セット方向への分割による占有領域（占有セット）の保持

トサイズが 3 セットであり，それぞれのコア上で動作するプロセスは，それぞれのコアが保持する占有セットにしかアクセスできない状況を表している．このように，各コア上で動作しているプロセスがアクセス可能なセットは限定されるため，干渉による性能低下を回避することができる．このことを確認するために 3.1 節で述べた，ストリーミング処理を行うプロセスにより干渉が発生し，性能が低下するような場合の提案手法の動作を説明する．これにあたり 3.1 節で用いた，一度しか参照されないアドレスへのアクセスが多発するストリーミング処理を実行するプロセスがコア A 上で，Dirty 領域に対する読み出しアクセスが多いプロセスがコア B 上で同時に動作している例を考える．この場合，提案手法では各コアが占有領域を保持しているため，たとえコア A 上で動作するプロセスが頻繁にエントリを確保しても，コア B 上で動作するプロセスが確保したエントリには影響を与えない．このため，RWP の場合に問題となっていた，コア B 上で動作するプロセスが確保したエントリを，コア A 上で動作しているプロセスがキャッシュメモリから追い出してしまう状況を回避できる．これにより，コア B 上で動作するプロセスの特徴を考慮して Dirty 領域の内訳を大きくした場合でもキャッシュ領域を十分に活用できる．このように，提案手法では干渉による性能低下に対処することができ，キャッシュ性能の向上が見込める．

さらに，この場合，コア A 上で動作しているプロセスの実行フェーズは，確保したキャッシュエントリをほとんど再参照しない実行フェーズである．したがって，このプロセスが必要とするキャッシュサイズは少ないと考えられ，このプロセスを動作させているコアが保持する占有領域が縮小したとしてもキャッシュ性能に影響を与えないと考えられる．また，コア B 上で動作しているプロセスはキャッシュエントリを頻繁に再参照しており，

保持する占有領域の拡大に伴いキャッシュ性能が向上する見込みがあると考えられる．このため，コア A の保持する占有領域を縮小し，その縮小分をコア B の保持する占有領域に割り当てて拡大し，コア B で動作するプロセスが利用可能となるキャッシュサイズを増加させることにより，さらにキャッシュ性能を向上させることができると考えられる．そこで，提案手法では，各コア上で動作しているプロセスの読み出しアクセスのミス回数を記憶し，この回数に基づいてコア毎の占有領域のサイズを調整することでキャッシュ性能を向上させる．具体的には，最もミス回数の多いプロセスと最もミス回数の少ないプロセスを検出し，後者のプロセスが動作しているコアが保持する占有領域を縮小し，前者のプロセスが動作しているコアが保持する占有領域を拡大する．このように，提案手法ではプロセス毎に必要とするキャッシュサイズが異なることを考慮して，占有領域サイズを動的に調整し，更なるキャッシュ性能の向上を図る．

さて，前述したように，セット方向へキャッシュ領域を分割する手法を RWP に適用したことで，各コアが占有領域を保持することが可能となる．このため，占有領域毎に予測 Clean/Dirty 内訳サイズを算出することで，異なる Clean/Dirty 領域の内訳サイズを占有領域毎に設定することが可能となる．ここで，RWP と提案手法の予測 Clean/Dirty 内訳サイズの算出に関する動作の違いを，3.1 節の場合と同様に，Clean 領域内のエントリに対する読み出しアクセスが頻繁に発生するプロセス（プロセス X とする）と，Dirty 領域内のエントリに対する読み出しアクセスが頻繁に発生するプロセス（プロセス Y とする）が同時に実行している場合を例に，**図 8** を用いて説明する．図 8 では，2 コア構成のマルチコア環境に対して，RWP と提案手法のそれぞれを実装した際の Age Hit Counter の様子を比較して示している．なお図 8 の (a) が RWP の実装，(b) が提案手法の実装を示している．プロセス X は Clean 領域を，プロセス Y は Dirty 領域をより多く必要とするため，両プロセスがより多く必要とする領域の種類は異なっている．しかし，図 8(a) に示すように RWP の場合は，Age Hit Counter が全体で 1 つしか存在せず，Age Hit Counter の値に基づいて算出する値である予測 Clean/Dirty 内訳サイズは，1 つしか求めることができない．よって RWP の場合は，異なるプロセスが同時に動作していても，一意に決定した予測 Clean/Dirty 内訳サイズに実際の Clean/Dirty 領域の内訳サイズが近づくように，キャッシュエントリを管理するため，各プロセスは本来必要とするサイズの Clean/Dirty 領域を十分に利用できない．実際に (a) に示すように RWP の場合は，キャッシュセット全体に対して，予測 Clean/Dirty 内訳サイズは 2 : 2 と算出され，プロセス X とプロセス Y がより多く必要とする領域の種類が Clean 領域と Dirty 領域とで異なっていることが考慮されていない．一方，図 8(b) に示すように提案手法では，占有領域の個数と同数

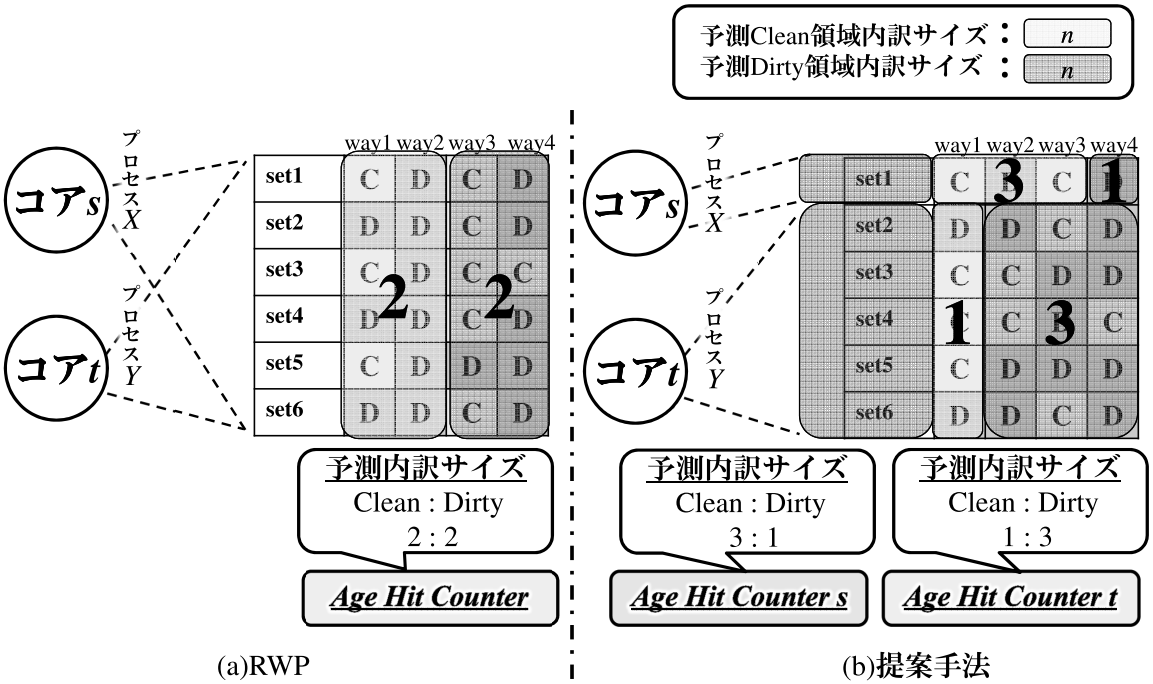


図 8: RWP と提案手法との Age Hit Counter の実装の比較

の Age Hit Counter を用意する．これは，エントリに対する読み出しアクセスのヒット回数をそのエントリが属する占有領域毎に記憶するためである．コア s が保持する占有領域に属するエントリに対しての読み出しアクセスのヒット回数を記憶するのが Age Hit Counter s，コア t が保持する占有領域に属するエントリに対する同様の回数を記憶するのが Age Hit Counter t である．これにより，プロセスを動作させているコアが保持する占有領域毎に，異なる予測 Clean/Dirty 内訳サイズに実際の Clean/Dirty 領域の内訳サイズが近づくようにキャッシュエントリを管理することが可能となる．実際に (b) に示す提案手法の場合は，プロセス X が動作しているコア s が保持する占有領域に対しての予測 Clean/Dirty 内訳サイズは，Age Hit Counter s の値に基づいて，3 : 1 と算出され，プロセス Y が動作しているコア t が保持する占有領域に対しての予測 Clean/Dirty 内訳サイズは，Age Hit Counter t の値に基づいて，1 : 3 と算出される．このように，各プロセスがより多く必要とする領域の種類に合わせた予測 Clean/Dirty 内訳サイズを算出可能である．このため，RWP の場合のように，Clean 領域を多く必要とするプロセスが，十分な大きさの Clean 領域を利用できないといったような状況を回避できると考えられる．以上のことから，提案手法ではプロセスの実行フェーズに応じた Clean/Dirty 領域の分割だけでなく，プロセス毎の異なる特徴に応じた Clean/Dirty 領域の分割を行うことも可能となり，より一層のキャッシュ性能の向上が見込める．

4　提案手法の実装と動作

本章では，3.3 節で述べた提案手法を実現するために必要となる機能と，それらの機能を実現するハードウェア機構について述べる．そして，これらのハードウェア機構を用いた動作を，提案手法により実現可能となる機能毎に説明する．

4.1　追加ハードウェア

3.3 節で述べた提案手法を実現するためには，以下に示す機能を RWP に追加する必要がある．

- より多くのキャッシュサイズを必要とするプロセスを動作させているコアが，より多くの占有領域を保持できる機能
- どのコアが保持している占有領域に各セットが含まれているかを，キャッシュエントリ確保時に判断できる機能

1 つ目の機能は 3.3 節で述べた，プロセス毎の必要なキャッシュサイズに応じた占有領域サイズを設定するために必要な機能である．2 つ目の機能は，占有領域を管理するために必要な機能である．提案手法では，これらの機能を実現するために，コア毎の読み出しアクセスのミス回数を記憶するハードウェア（**Core Counter**）と，各セットが含まれている占有領域がどのコアに保持されているかを記憶するハードウェア（**Core Directory**）を追加する．Core Counter は実装する環境の総コア数と同数のエントリを持ち，各エントリは以下の 2 つのフィールドを持つ．

core_number ： 各コアのインデックス番号，つまりコア番号を記憶する

read_miss_count ： 各コアの読み出しアクセスのミス回数を記憶する

なお read_miss_count フィールドの値は，一定間隔でリセットされる．これはプロセスが必要とするキャッシュサイズが実行フェーズに応じて変化することに対応するためである．また，Core Directory は実装する環境の共有セットのセット総数と同数のエントリを持ち，各エントリは以下の 2 つのフィールドを持つ．

set_number ： 各セットのインデックス番号，つまりセット番号を記憶する

partition ： set_number の値と対応するセットが含まれている占有領域を保持しているコアのインデックス番号，つまりコア番号を記憶する

このほかに，提案手法では，占有領域毎にエントリに対する読み出しアクセスのヒット回数を記憶するためのハードウェア機構である，Age Hit Counter をコア数分用意する．これにより，占有領域毎に予測 Clean/Dirty 内訳サイズを算出することができる．以上で

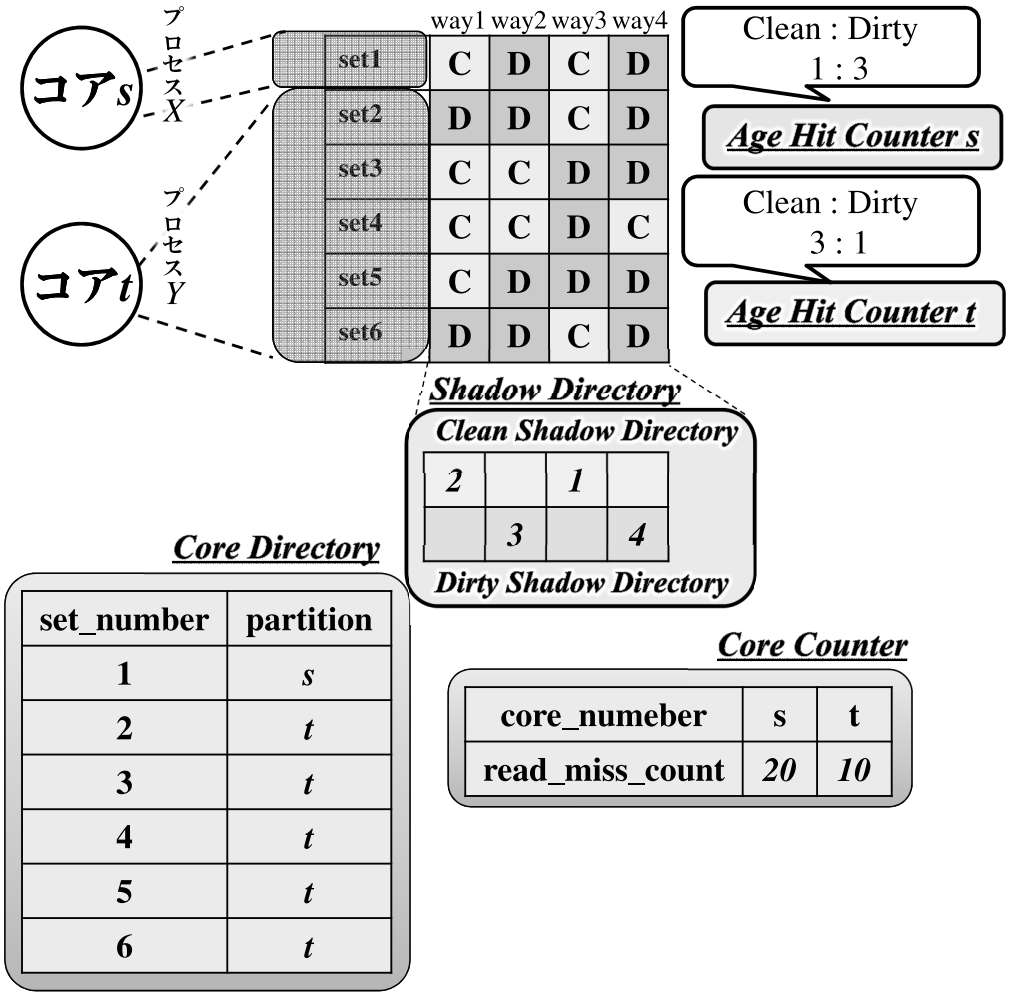


図 9: 提案手法に必要なハードウェア追加後の様子（2 コア）

述べた追加ハードウェアを 2 コア構成のマルチコア環境下に実装した際の，ハードウェア構成を**図 9** に示す．なお，説明の簡単化のため，共有キャッシュが持つセットの総数は 6 セットと仮定している．このため図 9 に示すように，Core Directory は 6 つのエントリを持つ．また，2 コア構成のマルチコア環境であるため，Core Counter は 2 つのエントリを持ち，共有キャッシュはセット方向に 2 つの領域に分割され，その領域を 2 つのコアがそれぞれ占有領域として保持している．そして，キャッシュエントリに対する読み出しアクセスのヒット回数は，そのアクセスがコア s 上で動作しているプロセス X によるアクセスであれば，コア s 用の Age Hit Counter s に，コア t 上で動作しているプロセス Y によるアクセスであれば，コア t 用の Age Hit Counter t にそれぞれ記憶される．したがって，予測 Clean/Dirty 内訳サイズは，プロセス X，プロセス Y のそれぞれに対して Age Hit Counter s，Age Hit Counter t の値に基づいて算出される．提案手法では，以上で

述べた追加ハードウェアを用いることで，各コアに占有領域を与え，干渉による性能低下を回避する．また，プロセスの実行フェーズを考慮した占有領域サイズの動的調整を実現し，さらにプロセス毎の特徴を考慮した Clean/Dirty 領域の分割を可能にする．なお，以上で述べた追加ハードウェアのコストについては 5.4 節で考察する．

4.2 占有領域保持による干渉の発生回避

提案手法では，各コアが占有領域を保持することで，干渉の発生を回避することが可能となる．また，この占有領域のサイズを動的に調整することも可能となる．本節では，この占有領域のサイズが動的に変化する際に，提案手法がどのように動作するのかを説明する．提案手法では，Age Hit Counter の値をリセットする際に，占有領域サイズを動的に調整する．この調整には前述したコア毎の読み出しアクセスのミス回数を記憶している Core Counter と，各セットが含まれている占有領域がどのコアに保持されているかを記憶している Core Directory を用いる．まず，どのコアが保持する占有領域を拡大/縮小するべきかを判断するために，Core Counter を参照する．具体的には，最もミス回数の多いコアが持つ占有領域を拡大し，最もミス回数の少ないコアが持つ占有領域を縮小するべきと判断する．例えば，Core Counter が**図 10** に示す状態の場合，最もミス回数の多いコアはコア s，最もミス回数の少ないコアはコア t である．よって，コア t が保持する占有領域を縮小し，コア s が保持する占有領域を拡大するべきと判断する．その後，各セットが属している占有領域を変更したことを記憶するために Core Directory を更新する．これは，縮小する占有セット数と同数の Core Directory 内の各エントリに対して，そのエントリが持つ partition フィールドの値を，拡大する占有領域を保持しているコアのインデックス番号に書き換え，実現する．例えば，Core Directory が**図 11** に示す状態かつ，Core Counter が図 10 に示す状態で，コア t が保持する占有セットを 2 セット分縮小する場合，各エントリの partition フィールドの値がコア t のインデックス番号と一致しているエントリを探索し，そのエントリの partition フィールドを s に変更するという操作を 2 回行う．このようにして，提案手法では占有領域サイズを動的に調整する．

また，キャッシュに対するアクセスがミスし，新たにエントリが確保される際に，そのアクセス要求を出したプロセスを動作させているコアが保持する占有領域以外に，エントリが確保されることがあってはならない．このため，新たなエントリは，アクセスにミスしたプロセスが動作しているコアが保持する占有領域を Core Directory に基づいて判断し，その占有領域内に確保されるようにハッシュ関数により調整する．例えば，Core Directory が図 11 のような状態の場合に，コア s で動作するプロセスのアクセスがミスし，

Core Counter

core_number	1	2	…	*s*	…	*t*	…	*N*
read_miss_count	*20*	*10*		*32*		*2*		*8*

図 10: Core Counter

Core Directory

set_number	partition
1	*s*
2	*t*
3	*t*
4	*t*
5	*t*
6	*t*

s に変更

図 11: Core Directory

新たなエントリを確保する必要が生じた場合について考える．この場合，Core Directory からコア s が保持する占有領域はセット 1 のみであることが分かる．したがって，新たに確保されるエントリはセット 1 中に確保されるようにハッシュ関数により調整される．以上のように動作することで，提案手法では干渉の発生を回避し，占有領域サイズの動的調整を実現する．

4.3 占有領域毎の Clean/Dirty 分割

提案手法では，予測 Clean/Dirty 内訳サイズは各コアの保持する占有領域毎に算出される．本節では，キャッシュメモリに対する読み出しアクセスがヒットした場合と，キャッシュメモリに対するアクセスがミスした場合に分けて，その予測 Clean/Dirty 内訳サイズの算出に関連する提案手法の動作を説明する．この説明には，2 コアのマルチコア環境に対して提案手法を実装した様子を示している**図 12** を用いる．なお図 12 はプロセス A，プロセス B がそれぞれコア s，コア t 上で動作し，これらのコアが共有キャッシュ上に占有領域を保持している状況を表している．

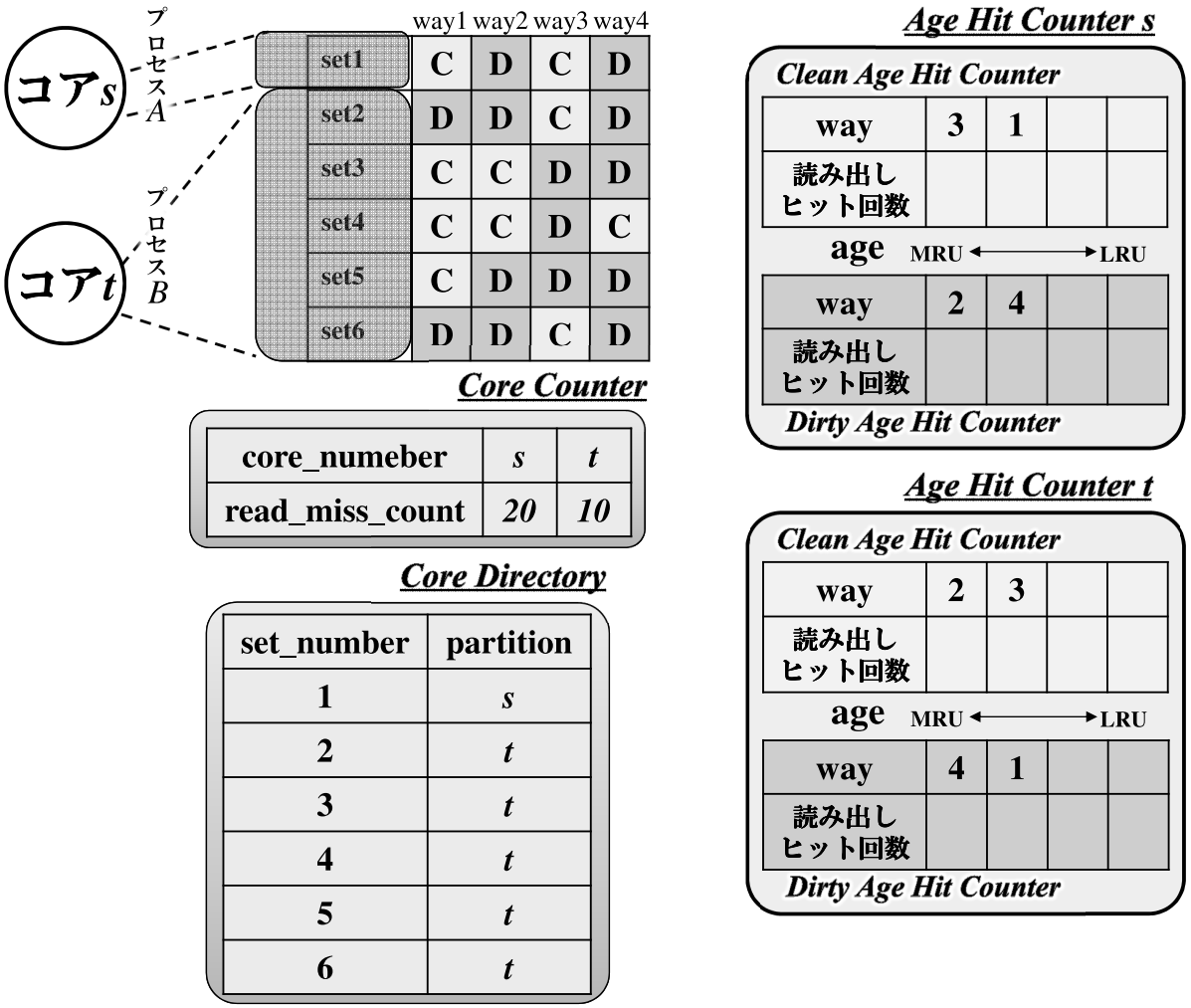


図 12: 占有領域毎に Age Hit Counter が存在（2 コア）

まず，キャッシュメモリに対する読み出しアクセスがヒットした場合の動作を説明する．この場合，各エントリに対する読み出しアクセスのヒット回数を占有領域毎に別の Age Hit Counter を用いて記憶する必要がある．そのため，もし読み出しにヒットしたエントリが，Shadow Directory で管理されているエントリであった場合は，そのエントリの属する占有領域に対応した Age Hit Counter の値をインクリメントする．この際，対応する Age Hit Counter がどれであるかは Core Directory を用いて判断する．ここで，これらの動作を具体的に説明するために，図 12 に示す状態で，プロセス B によるセット 3 に対する読み出しアクセスがヒットした場合を想定する．この場合，まず，読み出しにヒットしたエントリが属する占有領域を保持しているコアを判断するために，Core Directory 内のエントリから set_number フィールドの値がアクセス先セットのセット番号である 3

と一致するエントリを探し，その partition フィールドの値を参照する．そして，そのコアが保持している占有領域内のエントリに対する読み出しアクセスのヒット回数を記憶するための Age Hit Counter を見つける．図 12 の例では，partition フィールドの値は t であるため，セット 3 はコア t が保持する占有領域に属していることが分かる．よって，読み出しアクセスのヒット回数は Age Hit Counter t に記憶することになり，このアクセスが way2，つまり Clean な領域に属するエントリに対するアクセスであるならば，Age Hit Counter t 内の Clean Age Hit Counter の値がインクリメントされる．同様に，このアクセスが way4 のような Dirty な領域に属するエントリに対するアクセスであった場合は Age Hit Counter t 内の Dirty Age Hit Counter の値がインクリメントされる．提案手法ではこのようにして占有領域毎に別の Age Hit Counter を用いて，エントリに対する読み出しアクセスのヒット回数を記憶する．

次に，キャッシュメモリに対してのアクセスがミスし，Clean/Dirty のどちらの領域からエントリを追い出すかを決定する際の動作を説明する．この場合，アクセスにミスしたプロセスを動作させているコアが保持する占有領域を考慮して予測 Clean/Dirty 内訳サイズを求める必要がある．このため，予測 Clean/Dirty 内訳サイズは，アクセス先のエントリが含まれている占有領域に対応する Age Hit Counter の値に基づいて算出する．この際，この対応する Age Hit Counter がどれであるかは，読み出しアクセスがヒットした場合と同様に Core Directory を用いて判断する．ここで，図 12 に示す状態で，プロセス B によるアクセスがミスした場合を想定する．この場合，もし，アクセス先セットが 2 であったとすると，まずそのセットの含まれる占有領域を保持しているコアを判断するために，Core Directory 内のエントリから，set_number フィールドの値がアクセス先セットのセット番号である 2 と一致するエントリを探し，その partition フィールドの値を参照する．図 12 に示すように，partition フィールドの値は t であるため，セット 2 はコア t が保持する占有領域に属していることが分かる．したがって，予測 Clean/Dirty 内訳サイズは，コア t が保持する占有領域内のエントリに対する読み出しアクセスのヒット回数を記憶している Age Hit Counter t の値に基づいて算出すべきと判断される．提案手法では，このようにして予測 Clean/Dirty 内訳サイズを算出するために用いる Age Hit Counter を選択する．その後，2.2.3 項で説明した既存の RWP と同様の方法を用いて，選択された Age Hit Counter の値に基づいて予測 Clean/Dirty 内訳サイズを算出する．各 Age Hit Counter にはキャッシュメモリに対する読み出しアクセスがヒットした際に，そのアクセス先エントリが属している占有領域に対応して，読み出しアクセスのヒット回数が記憶されているため，この算出した値は，アクセス先エントリが属している占有領域を保持して

いるコア上で動作しているプロセスの特徴を反映した値となる．そして，この算出した予測 Clean/Dirty 内訳サイズに近づくように，エントリをキャッシュメモリから追い出し，実際の Clean/Dirty 領域の内訳サイズを調整する．以上のように動作することで，提案手法では占有領域毎に予測 Clean/Dirty 内訳サイズを算出し，各占有領域を保持するコア上で動作するプロセスの持つメモリアクセスの特徴を考慮した Clean/Dirty 領域の分割を実現する．

5　評価

前章までに述べた，提案手法の有効性をシミュレーションにより評価し検証した．本章では，評価に用いたワークロード及び評価環境について述べた後，得られた評価結果を示し，それを考察する．そして，提案した手法を実現するために必要なハードウェアコストを見積り，それを考察する．

5.1　評価方法

ワークロードの作成方法は，RWP[9] の評価と同様の方法を選択した．この方法では，まず SPEC CPU 2006 の 29 プログラムをメモリインテンシブかどうかで 2 分類する．そして，(A) メモリインテンシブなプログラム同士の組み合わせで作成するワークロードと，(B) メモリインテンシブでないプログラムを含む組み合わせで作成するワークロード，(C) メモリインテンシブでないプログラム同士の組み合わせで作成するワークロードを用意する．文献 [9] によるとメモリインテンシブなプログラムは**表 1** に示した 10 種類のプログラムである．RWP の評価方法にならい，(A) の作成には 429.mcf，434.zeusmp，456.hmmer，462.libquantum，482.sphinx3 を用いた．そして，(B)，(C) の作成には全ベンチマークプログラム 29 種類を用い，(A) と重複しないように組み合わせを考慮し，ワークロードを 30 個作成した．

なお，**表 2** に示すように SPEC CPU 2006 の 29 種類のプログラムは各プログラムの特徴に基づき以下の 4 つに分類することができる．

- Insensitive
  キャッシュサイズによって殆ど性能が変化しない
- Cache-friendly
  キャッシュサイズを大きくするほど性能が向上する
- Cache-fitting
  キャッシュサイズを大きくすると，ある一定のサイズでミスがほぼなくなる

表 1: メモリインテンシブなプログラム

メモリインテンシブ	400.perlbench, 429.mcf, 434.zeusmp, 435.gromacs, 437.leslie3d, 450.soplex, 471.omnetpp, 481.wrf, 482.sphinx3, 483.xalancbmk

表 2: SPEC CPU 2006 のベンチマークの分類

Insensitive	400.perlbench, 410.bwaves, 416.gamess, 435.gromacs, 444.namd, 445.gobmk, 447.dealII, 453.povray, 454.calculix, 456.hmmer, 458.sjeng, 464.h264ref, 465.tonto, 481.wrf
Cache-friendly	401.bzip2, 403.gcc, 434.zeusmp, 436.cactusADM, 437.leslie3d, 473.astar
Cache-fitting	450.soplex, 470.lbm, 471.omnetpp, 482.sphinx3, 483.xalancbmk
Thrashing/Streaming	429.mcf, 433.milc, 459.GemsFDTD, 462.libquantum

- Thrashing/Streaming
  キャッシュサイズを大きくしてもミス数が非常に多い

(B)，(C) の作成にはこの分類も考慮し，各分類の組み合わせに偏りが生じないようにした．以上のようにしてワークロードを計 35 個用意し，提案手法の性能を評価した．

5.2 評価環境

シミュレータには鬼斬弐 [15] を利用した．この評価に用いたシミュレータ仕様を**表 3** に示す．また，コア数は 2 とし，キャッシュ構成は RWP にならい，L3 キャッシュのサイズを 4Mbytes に設定した．なお，提案手法において各コアが保持する占有領域のサイズは，実装の簡単化のためキャッシュセットの総数を 32 分割した単位で変化するものとし，提案手法は L3 キャッシュにのみ適用した．また，占有領域サイズの決定頻度は 5M 命令毎とした．評価対象のプログラムとしては，5.1 節で述べたように，SPEC CPU 2006 の全 29 プログラムを選択し，その入力データセットとしては最もサイズの大きい referenced を利用した．そして，5.1 節で述べたワークロードを用いて，250M 命令のスキップ後に，各プロセスの実行した命令数が合わせて 500M 命令となるまでをシミュレーションを行い，提案手法の性能を評価した．なお，今回は事前評価として，作成した 35 個のワーク

表 3: シミュレータ諸元

Processor	
ISA	Alpha
number of cores	2 cores
issue order	out-of-order
issue width	int:2, fp:2, mem:2
instruction window	int:32, fp:16, mem:16
branch pred	8KB g-share
BTB	2K entries, 4 ways
RAS	8 entries
L1 I&D cache	32 KBytes
ways	8 ways
latency	2 cycle
line size	64 Bytes
L2 cache	256 KBytes
ways	8 ways
latency	8 cycles
line size	64 Bytes
L3 cache	4 MBytes
ways	16 ways
latency	30 cycles
line size	64 Bytes
Memory	infinity
latency	200 cycles

ロードの内の 5 個のみを評価に用いた.

5.3 性能評価の結果と考察

3 章および 4 章で述べた提案手法が，RWP と比較して性能向上を得られるかどうかについて検証するため，キャッシュエントリの管理に単純に追い出しアルゴリズムとして LRU を用いた場合，RWP を用いた場合，提案手法を用いた場合の 3 通りについて評価した．各ワークロードを用いて評価した結果を**図 13** に示す．評価結果は，評価に用いた

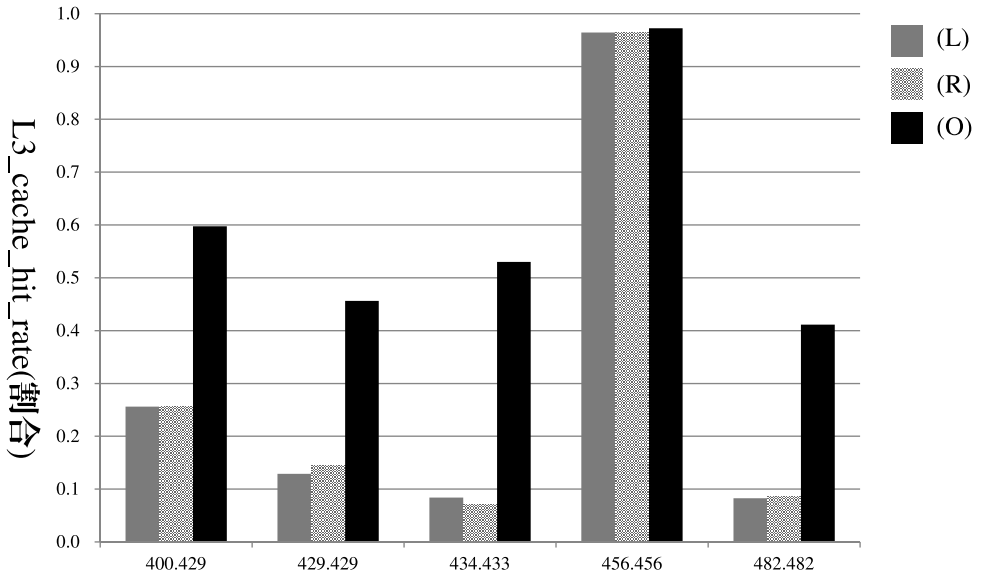


図 13: 共有キャッシュに対する読み出しアクセスのヒット率

ワークロード毎にそれぞれ 3 本のバーで示しており，それぞれのバーは左から順にキャッシュエントリの管理に

- (L) 追い出しアルゴリズムとして LRU のみを用いた場合
- (R) RWP を用いた場合
- (O) 提案手法を用いた場合

の性能を表しており，図 13 は各ワークロードを用いて評価した場合の共有キャッシュに対する読み出しアクセスのヒット率を示している．

キャッシュエントリの管理に提案手法を用いた場合である (O) では，すべてのワークロードにおいて読み出しアクセスのヒット率が向上している．ここで，評価に用いたワークロードは，ストリーミング処理やスラッシング処理を行うベンチマークプログラムを含んでおり，提案手法による効果が大きくなると考えられるワークロードと，そうでないワークロードに 2 分類することができる．まず，前者のストリーミング処理やスラッシング処理を行うベンチマークプログラムを含んだワークロードを用いた場合に，大きな性能向上率が得られた理由を考察する．なお，これに該当するワークロードは表 2 に示すように，429.mcf，433.milc を含むワークロードである．これらのワークロードを含む場合は，3.3 節で述べたように干渉が頻繁に発生してしまうことにより，キャッシュの性能が著しく低下してしまう場合がある．これが (L)，(R) において読み出しアクセスのヒット率が低くなっている原因であると考えられる．一方で (O) では，各コアが占有領域を持つため干渉が発生することはない．したがって，高いヒット率を達成することができたと考えら

れる．次に，ストリーミング処理やスラッシング処理を行うベンチマークプログラムを含まないワークロードを用いた場合にヒット率が向上した理由を考察する．これに該当するワークロードは，482.sphinx3 同士，および 456.hmmer 同士の組み合わせで作成したワークロードである．まず，482.sphinx3 は表 1，表 2 より，メモリインテンシブなプログラムかつ，Cache-fitting なプログラムである．このため，キャッシュを共有していた場合には干渉が発生してしまうことにより，各コア上で動作しているプロセスが必要とするサイズのデータがキャッシュ上に収まることがなく，ヒット率がそれほど高くなることは無かった．しかし，提案手法により各コアが占有領域を持ったことで，干渉が発生しなくなり，プロセスが必要とするサイズのデータがキャッシュに収まるようになったためヒット率が向上したと考えられる．一方で，456.hmmer 同士の組み合わせで作成したワークロードを用いた場合にヒット率がほとんど変化しなかったのは，単純に (L)，(R) における読み出しアクセスのヒット率が十分高く，向上の余地が少なかったためである．(L)，(R) において読み出しアクセスのヒット率が高くなるワークロードは多く存在しているはずであるが，今回の事前評価に用いたワークロードにはストリーミング処理やスラッシングを行うベンチマークを含んだものを多く選択した．そのため，このようなワークロードは 1 つしか確認することができなかった．そこで，今後さらに評価するワークロードを増やし，考察する予定である．

5.4 ハードウェアコストの見積り

本節では，これまでに提案した手法を実現するために必要なハードウェアコストを見積り，それについて考察する．Core Counter の全エントリ数は実装する環境のコア数と同数となり，各エントリは前述したように core_number と read_miss_count の 2 つのフィールドを持つ．このフィールドはそれぞれ $\log_2\{$ コア数 $\}$ $[bits]$，$\lceil \log_2\{$ カウンタリセット間隔命令数 $\} \rceil$ $[bits]$ の幅が必要となる．このため，Core Counter 全体では $\{$ コア数 $\} \times \log_2\{$ コア数 $\} + \lceil \log_2\{$ カウンタリセット間隔命令数 $\} \rceil$ $[bits]$ 必要となる．仮に，コア数が 2 で，カウンタリセット間隔命令数が 5M だとすると，core_number フィールドに必要なビット幅は $1[bit]$ で，read_miss_count フィールドに必要なビット幅は $2^{23} = 8{,}388{,}608 > 5M$ より，$23[bits]$ となる．したがって，Core Counter の 1 エントリに必要なビット幅は $24[bits]$ となり，全体では，$2 \times 24 = 48[bits] = 6[Bytes]$ 必要となる．

また，Core Directory の全エントリ数はキャッシュセットの総数と同数となり，各エントリは前述したように set_number と partition の 2 つのフィールドを持つ．このフィールドはそれぞれ $\log_2\{$ 総セット数 $\}$ $[bits]$，$\log_2\{$ コア数 $\}$ $[bits]$ の幅が必要となる．この

ため，Core Directory 全体では { 総セット数 } × ($\log_2${ 総セット数 } + $\log_2${ コア数 }) $[bits]$ 必要となる．仮に，コア数が 2 で，共有キャッシュの総セット数が 4,096 セットだとすると，set_number フィールドに必要なビット幅は 12$[bits]$ で，partition フィールドに必要なビット幅は 1$[bits]$ となる．したがって，Core Directory の 1 エントリに必要なビット幅は 13$[bits]$ となり，全体では 4,096 × 13 = 53,248 $[bits]$ = 6.656$[KBytes]$ 必要となる．

最後に，Age Hit Counter の追加に伴うコストの増加は，({ コア数 } − 1) × { Age Hit Counter1 個のコスト } となる．つまり，Age Hit Counter 1 個のコストが，384$[bits]$[9] であるため，({ コア数 } − 1) ×384$[bits]$ 必要となる．

これらより，仮にコア数2，カウンタリセット間隔命令数5M，共有キャッシュの総セット数が 4,096 である環境に実装すると，提案手法全体で必要となるハードウェアのコストは 48+53,248+384$[bits]$ = 6.710$[KBytes]$ となる．これは既存の共有キャッシュサイズ 4$[MBytes]$ と相対的に比較しても，非常に小さい増加量となることを確認できた．

6 おわりに

本論文では，キャッシュエントリの再利用性を考慮してキャッシュ領域を分割する RWP を改良し，各コアに占有領域を与えることで干渉による性能低下を抑制する動的キャッシュ領域分割手法を提案した．提案手法の有効性を検証するため，SPEC CPU 2006 ベンチマークを用いて，評価を行った．その結果，既存の RWP と比較して，共有キャッシュに対する読み出しアクセスのヒット率を最大 46 %，平均 29 %向上できることが確認できた．これにより，提案手法の有効性を確認することができた．

本研究の今後の課題として，以下の 2 つが挙げられる．1 つ目は，ハードウェアコストの削減である．本研究で用いた Core Directory はキャッシュ上の全セットの情報を管理することを想定しているが，占有領域サイズを動的に調整する際の拡大/縮小するセット数を大きくすることで，一部のセットの情報のみを管理することが可能となると考えられる．よって，この変動サイズについては，さまざまな値を設定した場合の性能を評価し，適切な値を探っていく必要があると考えている．

2 つ目は，様々な異なるアーキテクチャへ実装し，性能を評価することである．評価環境のパラメータの内，特にキャッシュサイズとコア数は本研究で提案した手法の性能に大きく影響を与えると考えられる．したがって，これらの組み合わせを変化させて性能を評価し，より提案手法が有効に働く組み合わせを深く考察していく必要があると考えられる．これはシミュレータの設定および，用いるシミュレータの変更により実現可能である

と考えている．

謝辞

本研究のために，多大な御尽力を頂き，ご指導を賜わった名古屋工業大学の津邑公暁准教授，松尾啓志教授，齋藤彰一准教授，松井俊浩准教授，梶岡慎輔助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた津邑研究室，松尾研究室，齋藤研究室および松井研究室の方々に感謝致します．特に，研究テーマが異なるのにもかかわらず拙い文章を解読し，洗練された文章へと導いてくれた先輩方にここに深く感謝致します．

参考文献

[1] Seznec, A.: A case for two-way skewed-associative caches, *Proc. 20th Annual Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 169–178 (1993).

[2] Qureshi, M. K., Thompshon, D. and Patt, Y. N.: The V-way Cache: Demand Based Associativity via Global Replacement, *Proc. 32nd Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 544–555 (2005).

[3] D.Lee, J.Choi, J.H.Kim, S.H.Now, S.L.Min, Y.Cho and C.S.Kim: LRFU:A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies, *IEEE Trans. on Computers*, Vol. 50, No. 12, pp. 1352–1361 (2001).

[4] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C. and Emer, J.: Adaptive insertion policies for hight performance caching, *Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 381–391 (2007).

[5] Jaleel, A., Theobald, K. B., Steely, S. C. and Emer, J.: High performance cache replacement using re-reference interval prediction (RRIP), *Proc. 37th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 60–71 (2010).

[6] Khan, S. M., Wang, Z. and Jimenez, D. A.: Decoupled dynamic cache segmentation, *Proc. 18th IEEE Symp. on High Performance Computer Architecture (HPCA)*, IEEE Computer Society, pp. 1–12 (2012).

[7] Collins, J. D., Z, H. W., Tullsen, D. M., Hughes, C., fong Lee, Y., Lavery, D. and Shen, J. P.: Speculative Precomputation:Long-range Prefetching of Delinquent Loads, *Proc. 28th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 14–25 (2001).

[8] Lai, A.-C., Fide, C. and Falsafi, B.: Dead-block prediction and dead-block cor-

relating prefetchers, *Proc. 28th Annual Int'l Symp. on Computer Architecture (ISCA)*, ACM, pp. 144–154 (2001).

[9] Khan, S., R.Alameldeen, A. and Wilkerson, C.: Improving Cache Performance by Exploiting Read-Write Disparity, *Proc. 20th International Symposium on High-Performance Computer Architecture (HPCA)*, ACM (2014).

[10] Qureshi, M. K., Lynch, D. N., Mutlu, O. and Patt, Y. N.: A Case for MLP-Aware Cache Replacement, *Proc. 33rd Annual international symposium on Computer Architecture (ISCA)*, IEEE Computer Society, pp. 167–178 (2006).

[11] Mattson, R., Gecsei, J., Slutz, D. and Traiger, I.: Evaluation techniques for storage hierarchies, IBM Systems Journal, Vol9, No2, pp. 78–117 (1970).

[12] Qureshi, M. K. and Patt, Y. N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, *Proc. 39th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, IEEE Computer Society, pp. 423–432 (2006).

[13] Dybdahl, H., Stenstrom, P. and Natvig, L.: A cache-partitioning aware replacement for chip multiprocessors, *Proc. 13th Int'l Conf. on High Performance Computing (HiPC)*, Springer-Verlag, pp. 22–34 (2006).

[14] 浅見公輔, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 命令グループごとのキャッシュパーティショニング, 先端的計算基盤システムシンポジウム (SACSIS), pp. 65–69 (2013).

[15] 塩谷亮太, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬弐」の設計と実装, 先進的計算基盤システムシンポジウム (SACSIS), pp. 120–121 (2009).